Public Disclosure Authorized

Public Disclosure Authorized

DISCUSSION PAPER

REVISED
Report No. DRD113

HIGH LEVEL MODELING SYSTEMS AND NONLINEAR PROGRAMMING

by

Anthony Brooke
Arne Drud
Alexander Meeraus

December 1984

Development Research Department
Economics and Research Staff
World Bank

The World Bank does not accept responsibility for the views expressed herein which are those of the author(s) and should not be attributed to the World Bank or to its affiliated organizations. The findings, interpretations, and conclusions are the results of research supported by the Bank; they do not necessarily represent official policy of the Bank. The designations employed, the presentation of material, and any maps used in this document are solely for the convenience of the reader and do not imply the expression of any opinion whatsoever on the part of the World Bank or its affiliates concerning the legal status of any country, territory, city, area, or of its authorities, or concerning the delimitations of its boundaries, or national affiliation.

HIGH LEVEL MODELING SYSTEMS AND NONLINEAR PROGRAMMING

By

Anthony Brooke,*
Arne Drud,**
Alexander Meeraus**

Development Research Department
World Bank

December 1984

* Consultant, The World Bank, supported by external research budget RPO 673-06

** Staff Members, The World Bank

ABSTRACT

The advent of high level modeling systems will make non-linear programming (NLP) more productive. This judgment is illustrated by a comparison of one such system, the General Algebraic Modeling System (GAMS) with the traditional approach. The linkages between GAMS and four NLP codes are discussed to demonstrate to code developers how more 'machine friendly' codes will also become more 'user friendly.'

HIGH LEVEL MODELING SYSTEMS AND NONLINEAR PROGRAMMING

TABLE OF CONTENTS

Abstract

1. Introduction....1
2. The Modeling Process....3
3. GAMS and the Modeling Process....8
4. The GAMS interface to NLP codes....14
5. High Level Modeling Systems and NLP Code Developers....21
6. Experience with GAMS....22

Appendix 1: GAMS/NLP Interfaces....24
- A. Interface with GRG2....24
- B. Interface with CONOPT....27
- C. Interface with NPSOL....28

References....30

1. Introduction

This paper examines the potential advantages of high level modeling systems for users and developers of nonlinear programming codes, using a particular high level system, the General Algebraic Modeling System (GAMS), as an example.

Although nonlinear programming (NLP) has been under development for many years it still has not gained the same acceptance as its cousin, linear programming (LP), for at least three reasons. First, linear models are conceptually easier to understand, and the potential user community for LP is therefore much greater than that for NLP. This is a fact that even the best algorithms and tools cannot change. Second, many model formulation and report writing tools are available for the user of LP, but not for NLP. As a result, NLP model formulation and report generation are time consuming and expensive -- often disproportionately so by comparison with solution costs. Finally, NLP codes are much less reliable than LP codes, making new applications of NLP risky. Most practitioners know that if an NLP model does not solve right away, it can probably be solved if the formulation, the solution algorithm, or the code is changed. With traditional technology, however, the costs of reformulation are high, and those of switching algorithms even higher. In summary, NLP has typically been characterized by low productivity as a result of slow progress, high expected costs, and high risks; managers of projects that could, in principle, use NLP have therefore in practice preferred less sophisticated methods.

We believe that the introduction of high level modeling systems will change this picture completely, making the tools available for model formulation and report writing with NLP as powerful as those available for LP;

as a result, the speed and cost of model formulation under the two systems will become comparable. A modeling system cannot change the reliability of a single NLP code. It can, however, change the reliability of NLP modeling by allowing the user to switch algorithm at no cost and to change model formulation very cheaply.

The sections which follow will provide detailed evidence to support the general thesis outlined above. Section 2 describes the modeling process and explains why current techniques are inadequate. Specifically, in addition to a general shortage of tools, this inadequacy stems from the fact that the input to NLP codes has usually been designed to be machine friendly and little has been done to make it user friendly; moreover, different codes use very different types of input. The position is further complicated by the possibility that one code prefers one type of mathematical model formulation, while another may prefer a different but mathematically equivalent formulation.

Sections 3-6 show how GAMS, a high level modeling system developed at the World Bank, tries to improve on this situation. GAMS breaks the input phase into two parts. The first part, described in Section 3, comprises the model formulation process itself and the definition of the associated data. GAMS uses a powerful and algorithm independent representation of model and data, which allows the user to define large and complex models and to test them for logical consistency in a very short time. The second part of the input phase, discussed in Section 4, is the translation of the user's model representation into the input format of the NLP code. This is done automatically by GAMS once the user has chosen a NLP code. GAMS knows the

input (and output) format of several NLP codes, and the user can switch from one to another by simply changing a code name.

Section 3 and 4 focus on the user's problems, and the benefits he/she can get from a high level modeling system. Section 5 turns to the gains accruing to the developers of NLP codes. These gains are of at least three kinds. First, the productivity increases derived from NLP modeling systems can increase the overall market for these systems' associated codes. In addition, use of a high level system permits the code developer to concentrate on the optimization algorithm, and to limit his involvement with the tedious parts of software development such as user friendly input/output and documentation. Finally, once an interface has been written for a new code, the availability of a large set of test models written in a high level language will make testing of this new code easier and more thorough. The penalty for these benefits is that the input to NLP codes needs to be standardized, so that interfaces can be standardized or at least built from standard components. Section 6 concludes the paper with a short description of experience to date with the GAMS modeling system.

2. The Modeling Process.

The process of building and using models, linear or nonlinear, is composed of five interrelated steps: model formulation, model implementation, model solution, report writing, and model documentation. The efficiency of the overall modeling process depends on the techniques used in each of the five steps, and on how well these techniques are integrated. We will try in the following paragraphs to cover each step and emphasize the links between steps.

A mathematician may think of a model simply as a set of variables tied together by constraints or equations. However, a set of data, and transformations that convert the data into model coefficients are just as important in practice. We define the model formulation process as that of defining the basic data set, defining the transformations of the data that create the coefficients of the model and defining the variables and the constraints. Note that our definition contains both a procedural or recursive part, the data transformations, and a non-procedural or simultaneous part, the variables and the constraints. It is important to realize that we are probably dealing with multidimensional quantities such as demand in different regions for different products, production of different products in different plants in different periods etc. Basic data, derived coefficients, variables, and equations can all be multidimensional. It is also important to realize that exceptions exist in all but the most trivial models: initial and terminal periods differ, some plants only produce some products, and so forth. This is equivalent to saying that the multidimensional quantities can be sparse. The model formulation process must therefore rely on techniques for defining data, data transformations, and variables and equations over multidimensional domains, subject to exceptions or sparsity. Our conventional algebraic notation with subscripts and summations is well suited for this purpose, and most models are initially defined using this notation.

The subsequent process of implementing the model on a computer can be broken down into several linked subprocesses: structuring and entering the basic data, programming the data transformations, selecting the solution code, and translating the model into the format required by the solution code. The translation step will usually involve writing a FORTRAN subroutine that can

compute the constraint values for given values of the variables. It sometimes involves writing subroutines for the derivatives of the constraints: for large models it may also involve specifying a sparsity pattern.

Using conventional techniques, the FORTRAN code will be written using a mixture of two approaches: the block approach and the single equation approach. The block approach exploits the multidimensional structure of the problem. The code will use multidimensional arrays and will be full of DO-loops, but it will also contain complicated constructs to handle the exceptions; these make the code difficult to read, verify, and modify. This approach was used with the first GAMS-NLP interfaces; further details are given in the appendix. The single equation approach eliminates all the complications of exceptions by treating each equation and variable within a block as an individual equation or variable. The code becomes very simple, but also very long, and all the structure of the underlying model vanishes. It is therefore just as difficult to read and verify as the block structured code, and just as difficult to modify. It is also much more expensive to compile.

Some NLP codes require more than a FORTRAN subroutine that evaluates constraints. They may also require a sparsity pattern and/or a subroutine that computes derivatives. This additional input is essentially a transformation of the information already built into the constraint subroutine. The user must thus provide the same information in two formats, and in a consistent manner.

Once the groundwork has been covered, the solution of the model itself is usually straightforward. It involves selecting some initial values and calling the NLP code. Difficulties arise only if the NLP code cannot

solve the model, something that happens more frequently than modelers like to admit. There are only three remedies. The simplest one is to try with another set of initial values, for example, ones computed from the basic data. The two other approaches are to try a different NLP code, or to change the model formulation into an equivalent one, by scaling rows or columns, for example, and to try to solve the new model with the old NLP code. Both these alternatives can be very expensive since they involve redoing a great deal of labor intensive programming in the model implementation phase. We have mentioned that failures are common, but we should also note that most problems can be solved by an experienced modeler with one of the existing codes after a few initial points and a few formulations have been tried; thus, solutions can be obtained reasonably reliably if we are prepared to spend time and money.

After completion of the solution phase, we are now ready to produce a set of reports. Report writing involves accessing solution values, often both primal and dual, performing transformations of them, and writing tables with the transformed values. The transformations will often involve the original data set and/or some of the coefficients, so the report software must have access to all this data. The conventional technique is to write special purpose software for report writing, using either a block approach or a single item approach.

The remaining phase, documentation, should ideally be carried out in parallel with the other four. We usually think of documentation in terms of two items, the documentation of the mathematics of the model, and the documentation of how this model has been implemented on a computer. Recent papers, e.g. Gass, [8], discuss the documentation aspects of modeling, using some very expensive modeling exercises as examples. These papers implicitly

assume that conventional techniques were used for implementing the model; under these circumstances, the issue of documentation becomes to a large extent a matter of documenting the FORTRAN code, and of setting up mechanisms to ensure that the mathematical model documentation and the program documentation are updated in parallel, i.e. that they represent the same model.

All the work involved in conventional NLP modeling, except the solution process itself, thus relies on software specially developed for the model at hand, and the modeling process is therefore very labor intensive. It becomes also very inflexible because there is no good way of rewriting code if necessary. In addition, there is a significant probability that the model defined through our FORTRAN code differs from the model we think we have defined because of undetected errors.

We believe that NLP modeling can become a general tool only if current technology can be made cheaper and more reliable. This requires that the modeler should only enter a certain piece of information into the computer once, and that he/she should enter it in a format that is natural to him/her. The computer should then transform information as needed. Such a procedure has several advantages: it improves the productivity of the modeler by avoiding repetitions, it eliminates many transformation errors caused by manual operation, and it makes documentation much easier. It also eliminates the documentation of FORTRAN code, and can make the computer input self documenting.

We call a computer program that accepts user friendly model definitions and performs all the necessary transformations to interface with a solution code a modeling system. The next section decribes one such modeling

system, GAMS, and show how it can solve a number of the problems outlined above.

3. GAMS and the Modeling Process

This section briefly describes some of the key features of the GAMS language. The description is intended to illustrate with examples some of the most important features of GAMS and how they can increase the productivity of modeling; the reader who is interested in further details is referred to Kendrick and Meeraus [11] and Meeraus [13].

Models become large not because they contain many different concepts, but because the concepts they describe are repetitive and defined over large sets, e.g. demand at many locations or production in many plants. It is therefore important that a modeling system has facilities for defining sets, and for defining data, data transformations, variables and equations over sets or cartesian products of sets. Below are two examples of GAMS set definitions.

```
SET FA  FACTORS OF PRODUCTION / LABOR, CAPITAL /
    SC PRODUCTION SECTORS     / AGRICULT, MANUFACT, SERVICES /;
```

FA and SC are the names of the sets, LABOR and CAPITAL, and AGRICULT, MANUFACT, and SERVICES are the elements of the two sets, and the strings FACTORS OF PRODUCTION and PRODUCTION SECTORS are (optional) documentation text.

Data can be entered in simple tabular form:

```
TABLE  FACIN(FA,SC)  FACTOR INCOME IN DIFFERENT SECTORS

                AGRICULT      MANUFACT      SERVICES
CAPITAL         1200          2900
LABOR           2000          1500          2300
```

Here the FA and SC following FACIN indicates that FACIN is defined over the domains of factors and sectors. GAMS will check that the row and column labels of the table are members of the proper sets, i.e. that there are no spelling errors, and it will check in later occurrences of FACIN for proper use of indices. Note that sets in GAMS are deliberately unordered; the row labels are ordered differently in the SET statement and the TABLE statement, but GAMS interprets the data by matching the labels and not the positions.

Once the basic data have been defined with SET and TABLE statements, it is easy to transform them into the coefficients needed in the model. This is done using assignment statements like those below. (A PARAMETER in GAMS is used to collect and name exogenous data values.)

```
PARAMETER   SECTIN(SC)     TOTAL INCOME IN EACH SECTOR
            FCSH(FA,SC)    FACTOR SHARES OF INCOME IN EACH SECTOR;
            SECTIN(SC) = SUM( FA , FACIN(FA,SC) );
            FCSH(FA,SC) = FACIN(FA,SC) / SECTIN(SC);
```

As well as these conventional algebraic operations, GAMS defines set operations such as unions and intersections, and operators over cartesian products like SUM, PROD, MIN, and MAX, making it easy to define aggregations and matrix products, and to perform other frequently used transformations.

Using the basic data and the derived coefficients we can now define the variables and equations that make up the model. In the example below,

output Q is modeled with a Cobb-Douglas production function (Q0 and TOTFACT are assumed to be parameters defined elsewhere).

```
VARIABLES  Q(SC)       PRODUCTION IN EACH SECTOR
           IN(FA,SC)   INPUT OF EACH FACTOR IN EACH SECTOR
           POSITIVE VARIABLES Q,IN

EQUATIONS  PRDF(SC)    PRODUCTION FUNCTION IN EACH SECTOR
           FACT(FA)    FACTOR AVAILABILITY;

PRDF(SC).. Q(SC) =E= Q0(SC) * PROD( FA , IN(FA,SC)**FCSH(FA,SC) );
FACT(FA).. SUM( SC , IN(FA,SC) ) =L= TOTFACT(FA);
```

The careful reader will have noticed that one of the sectors, 'SERVICES', did not use one of the factors, 'CAPITAL'. The corresponding variable, IN('CAPITAL','SERVICES'), could therefore be removed from this equation. This and other types of exceptions frequently occur in large models, and a consistent and convenient way of representing them is crucial in any modeling system.

In GAMS, exceptions are defined with a $ or 'such that' operator. Applied to the FACT equation above it yields

```
FACT(FA).. SUM( SC$FCSH(FA,SC) , IN(FA,SC) ) =L= TOTFACT(FA);
```

This means that the summation is carried out only over the elements of the set FA for which FCSH(FA,SC) is nonzero. The $-operator can also be applied to the domain over which an equation or an assignment statement is defined. It is not necessary to put an exception into equation PRDF since GAMS recognizes that the term IN('CAPITAL','SERVICES')**0 has the constant value of 1 and that IN('CAPITAL','SERVICES') can thus be removed from the equation PRDF('SERVICES').

Exceptions can also be handled through the use of subsets. Assume that only the goods produced by the agricultural and manufacturing sectors are exported. We can create a subset of the export sectors and define certain parameters, variables and equations over this subset:

```
SET SCE(SC) EXPORT SECTORS  / AGRICULT, MANUFACT /
PARAMETER   EXO(SCE)    BASIC EXPORT QUANTITY
            ETA(SCE)    EXPORT ELASTICITY
            PW(SCE)     WORLD PRICES FOR TRADED GOODS
VARIABLES   EX(SCE)     EXPORT FROM THE TRADED SECTORS
            PD(SC)      DOMESTIC PRICE OF GOODS BY SECTOR
EQUATIONS   EXD(SCE)    EXPORT DEMAND EQUATION FOR TRADED
            SECTORS;

EXD(SCE)..  EX(SCE) =E= EXO(SCE) * (PW(SCE)/PD(SCE)) **
            ETA(STC);
```

Notice that PD is defined over the set of all sectors, but only referenced over the export subset in the export demand equation.

A model in GAMS is defined as a collection of equations and the variables they contain. To start optimizing we simply state the objective, its direction, and the generic problem class. Particular codes can be selected using OPTION statements.

```
MODEL EXAMPLE / PRDF, FACT, EXD, other equation names /;
OPTION NLP = nlp-code;
SOLVE EXAMPLE USING NLP MINIMIZING objective;
```

Notice, that the definition of the model is completely independent of the solution code. The only place it is mentioned is in the OPTION statement above. This is very important; if the solution attempt fails with one algorithm, an easy switch to another is made by changing this one statement.

The execution of the SOLVE statement entails several verification and testing steps and then, if all is well, the model implementation and model solution steps mentioned in section 2. GAMS will first analyze the model symbolically and determine if it is a member of the class of models mentioned on the SOLVE statement. GAMS will then create the problem representation in the correct format, design a solution strategy, compile the created FORTRAN if necessary and transfer control to the NLP code. After the solution code has terminated, GAMS will read the output and translate it back into GAMS notation for storage in the GAMS data base. Normally the user will not see any output from the NLP code. Only if it fails in an unexpected way will code specific output be returned to the user.

The report writing step is straightforward. The solution values were saved in the GAMS database by the SOLVE statement, and they can now be used in further data manipulations, often using the DISPLAY statement to design and print tables. Four quantities are saved in the database for each variable: the level value, the lower bound, the upper bound, and the marginal value; four similar quantities are saved for each equation. The four values of variable IN can be referred to as IN.L, IN.LO, IN.UP, and IN.M, respectively.

References to the four quantities associated with a variable or an equation can be on either the right or left hand side of an assignment, i.e., IN.L can be used to compute values for a report after a model has been solved, but it can also be assigned a value, e.g.

```
IN.L(FA,SC) = FACIN(FA,SC) ;
```

Assignments of level values are used to define initial values for an optimization, and all the data manipulation machinery of GAMS is available.

It is therefore easy to compute good initial values and to try alternative sets of these values. The use of the current level values as initial values for an optimization has an added advantage: if we want to solve a second related model, GAMS will automatically use the optimal values from the first model as initial values for the second. In some cases it is useful to solve one or more relatively simple problems, perhaps linear, to find a good starting point. The use of current level values as inital values is also extremely useful if an NLP code cannot solve a model. It is easy to try to solve the model with another code, starting from the intermediate solution found by the first code. All the user has to do is to specify the name of the new code, and GAMS can take care of the rest. This can also be useful after reformulations of certain parts of a model. The values of the variables will automatically be used by GAMS as long as their names are the same.

Documentation is the final step in the modelling process. The only thing that has to be documented is the GAMS source input; all other parts of the model are temporary data structures and can be ignored. GAMS has been designed to encourage the modeler to document the model as it is developed. All sets, parameters, variables, and equations can have documentation texts associated with them, and the documentation is kept in the GAMS database and displayed each time they are printed. The GAMS source input can be used as the complete documentation of the model, especially if the modeler makes a conscientious effort to make source text clear and easy to understand. The development of a model in GAMS is similar to that of a scientific paper: several drafts are usually needed before the message has been presented in a clear and concise way to make it as easy as possible for the reader. Several models developed in the World Bank now use the GAMS input as the final

documentation and include it in the final reports, as in for example Brown et al, [2], Choksi et al, [3], and Kendrick et al [10].

4. The GAMS interface to NLP codes.

One of the design philosophies of GAMS is that models should be represented in an algorithmically independent way. This is, of course, useful only if links exist to many different optimization codes, and preferably to codes that are built around different principles. At present GAMS can communicate with four LP systems and four NLP codes, and more links, both generic and specific, will be added. So as to add these in an efficient and reliable way it is desirable to build on common components. In order to define some reasonable common components this section provides a summary overview of the codes we have already integrated, and describe the existing interfaces. More technical information on the interfaces can be found in the appendix.

Table 1 gives an overview of several characteristics of the four NLP codes that can be used with GAMS. The codes are GRG 2 (Lasdon et.al. [12]), NPSOL (Gill et.al., [9]), MINOS 5.0 (Murtagh and Saunders, [16]), and CONOPT (Drud, [7]). The remainder of this section provides additional information about the entries in the table and the interface components.

The execution of a SOLVE statement always begins with some common steps. GAMS checks that all sets, parameters, and equations used in the model have been defined and it checks that the model belongs to the class of models that can be handled by the particular NLP code. The restrictions are that equations must be differentiable functions of the variables and that variables cannot be binary. The user can disable the differentiability test by defining

Table 1: SOME CHARACTERISTICS OF THE FOUR NLP CODES CURRENTLY ACCESSIBLE THROUGH GAMS

Code	GRG2	CONOPT	NPSOL	MINOS
Characteristics:				
Algorithm Type: 1/	GRG	GRG	SQP	PAL
Jacobian representation:				
Sparse	No	Yes	No	Yes
Equations ordered	No	No	Yes	Yes
Variables ordered	No	No	No	Yes
Non linear elements distinguished	No	Yes	No 2/	No 2/
Subroutines:				
Constraints and objective separated	No	No	Yes	Yes
Linear terms in functions	Yes	3/	Yes	No
Numerical derivatives	Yes	Yes	No	Yes
Numerical derivatives computed sparse 4/	--	Yes	--	No
Name to index mappings 5/	--	Yes	--	No
Miscellaneous				
Stand-alone system	Yes	Yes	No	Yes
System can be modified	Yes	No	Yes	Yes
System controls memory, error recovery and file buffers	No	Yes	No	No

Notes:

1. The algorithm types are: GRG - Generalized Reduced Gradient, SQP - Sequential Quadratic Programming, and PAL - Projected Augmented Lagrangian.
2. NPSOL considers all Jacobian elements in the nonlinear equations nonlinear, and MINOS considers all Jacobian elements occuring in both nonlinear variables and nonlinear equations to be nonlinear.
3. Linear terms can either be included in the FORTRAN subroutine or the coefficient can be defined through the MPS file.
4. Numerical derivatives are computed 'sparse' if several columns of the Jacobian can be computed in one call of the constraint subroutine (see Coleman and More, [4]).
5. Both MINOS and CONOPT define a sparse Jacobian through an MPS-type file, i.e., through names of variables. The subroutines use variable indices. CONOPT allows explicit mappings from names to indices, MINOS does not.

the model as a discontinous NLP (DNLP), but this must be done explicitly and should only be undertaken by experienced users.

Almost all general purpose NLP codes define the nonlinearities of a problem through one or more subroutines that appear to the code as black boxes. The NLP code delivers one vector of variable values to the subroutine and expects one or more vectors with constraint and/or derivative values back. The inner functioning of the subroutine is of no concern to the optimizer and GAMS can do anything it finds convenient inside the subroutine.

The first interface built was the one to GRG2. GRG2 does not distinguish between linear and nonlinear equations or between linear and nonlinear variables; it does not need sparsity information but it can compute numerical derivatives. We decided to rely on the numerical derivatives computed by GRG2, and thus had only to create a FORTRAN subroutine that could compute constraint values for given values of the variables. The subroutine uses the block approach referred to in Section 2. The complications arising from exceptions and sparsity of multidimensional variables and equations are discussed in some detail in Section A of the appendix.

The second interface was to CONOPT. CONOPT is a sparse GRG code requiring an MPS-type file to describe the sparsity pattern of the Jacobian. Each Jacobian element must be classified as linear or nonlinear. The values of the linear entries can be entered in the MPS file or evaluated by subroutine calls. CONOPT can compute numerical derivatives and it takes advantages of sparsity in this computation using methods similar to those in Coleman and More [4], so the number of function evaluations needed to compute derivatives is comparable to the maximum number of nonlinear Jacobian entries in any row. Again, we used derivatives computed by differencing and decided

to include the linear components of the constraints in the FORTRAN subroutine. The interface then breaks down into the following tasks: write a FORTRAN subroutine to evaluate constraint values, and write an MPS file in which each Jacobian element is merely classified as linear or nonlinear. The code for the first task was derived from the GRG2 interface, but modifications were necessary to create the sparsity pattern and to check for exceptions caused by particular data values, such as multiplication by zero. The linearity or nonlinearity of Jacobian elements was determined on a block basis only, e.g., for all IN variables in all PRDF equations as one block, by analyzing the internal GAMS code. The technical aspects of the analysis are explained in Section B of the appendix.

The first two interfaces were built for the original machine specific CDC version of GAMS. This was designed for high performance on large linear models, and the NLP links were built using as many components of the LP framework as possible. The aim was to get quick and cheap experience with NLP GAMS. These links were very successful, and GAMS is currently used with more non-linear problems than linear ones. GAMS has now been rewritten to be machine independent, and the design of the new system pays much more attention to the needs of non-linear codes. In particular we decided to eliminate numerical derivatives. We no longer use block written code to specify non-linearities, but rather single equation specifications. We are also no longer writing FORTRAN but instead employ symbolic instructions which are interpreted (and which could easily be replaced by machine code for performance). NPSOL was chosen as the first code to be linked with the new system because it does not require sparsity information and is therefore easier to work with, and because it uses a sequential quadratic programming algorithm that has recently

been highly recommended. (Schitkowsky, [17]). There are three important differences between the requirements of NPSOL and the other fully allocated code, GRG2. First, NPSOL handles linear equations separately: they must be last and they must be defined by a matrix rather than a subroutine. Second, NPSOL cannot compute numerical derivatives. And third, the objective function must be written in a separate subroutine. The first difference is easy to handle. The LP component of GAMS can analyze equations for linearity and it finds the values of the derivatives in the same way as in a matrix generator. The difference due to the separate treatment of the objective is essentially a bookkeeping problem and is also easily handled. We are therefore left with only one new difficulty in this interface, the derivatives. We have implemented a routine that creates internal GAMS instructions for the derivatives based on the principle of point derivatives and on repeated use of the chain rule. This code is then evaluated in the same way as the code for the constraints. The point derivatives take full advantage of intermediate values from the constraint evaluation and the resulting code is therefore quite efficient. It is more complicated to create derivatives in block mode because of the possible exceptions involved; this was one of the reasons for experimenting with a single equation approach. We expect that GAMS will ultimately decide automatically whether to use block or single equation codes, and that it will switch modes depending on the problem structure. Technical details can be found in section C of the appendix.

The most recent link is between GAMS and MINOS 5.0. MINOS is more demanding than the other codes. It requires that variables be sorted so that nonlinear ones appear first, and that the FORTRAN subroutine should compute only those parts of the nonlinear constraints that depend on the nonlinear

variables. The order of variables is no problem. GAMS uses mapping vectors to get from the vector of variables in the NLP code to the variables as GAMS sees them, and this mapping vector can sort the variables in any order. The split of constraint elements into a linear and a nonlinear partition is more difficult. It requires two passes over the model because the classification of a variable as linear or nonlinear is global to the model and not local to the equation. The first pass determines which variables are nonlinear and the second removes the terms that depend on linear variables. Although MINOS can compute numerical derivatives we decided not to used this feature, but used point derivatives as in NPSOL.

The preceding paragraphs have discussed the main components of the interfaces, the generation of nonlinear function values and derivatives, and the sorting and partitioning of data structures. The remaining questions are how to move initial values from GAMS into the NLP code, how to move solution values back again, how to take care of error recovery, and how to pass control between the two systems.

Moving initial values from GAMS to GRG2, CONOPT, and NPSOL is straightforward. All these codes simply receive a vector of variable values in their ordinary input and start from there. MINOS is slightly more complicated, since the variables must be classified as basic, superbasic, or nonbasic at either bound. Moving information back is more systems specific. For the codes using block written FORTRAN (GRG2 and CONOPT) GAMS does not know which variables have been eliminated. The solution is therefore written back in the same partial order in which GAMS presented the rows and columns to the code. By encoding name strings GAMS is then able to associate each solution record with the appropriate entry in the GAMS data base. With NPSOL and MINOS

name matching is not needed because the loops have been unrolled by GAMS itself and all empty single rows or columns have already been eliminated.

We have taken great care to classify all possible termination states that are detected by GAMS when reading the output file from the specific NLP system. We all know the disappointment when an output says "Division by zero - Abort"; we want to make sure that this type of message is not seen by the modeler. First, GAMS makes sure that a syntactically incorrect model is never passed to the NLP code. Second, we have assured that the subroutines GAMS creates cannot cause an abort. We check for division by zero and recover from errors in mathematical functions caused by undefined operations. These execution time errors are then reported back to the modeler with information on where in the model the error occurred; the modeler can then decide if the model is wrong, or if the error was simply caused by the lack of some appropriate bounds. Execution errors in the NLP code itself are reported as such with an encouragement to submit the example to the code developer for further study.

Some of the NLP codes are set up as subroutines, and they could in principle be called directly from GAMS using what Drud [16] called an internal interface. This possibility is not available in all cases, however, notably not with commercial LP systems. We have therefore decided to use what we call an external interface. GAMS creates all the necessary information in a set of files and calls the NLP code through a procedure at the operating system level. (In a multitasking environment we could spin off a subtask and let GAMS wait for its completion.) The last task of the procedure is to give control back to GAMS, and GAMS can then read the output file from the NLP code. Apart from the fact that this approach is the only one available in

some cases, it has several advantages. It limits the use of memory (unless overlay structures are used), and only the code that is actually used is loaded. More importantly, this approach makes it very clear where errors occur, and consequently makes it easier to find them. The intermediate files can be captured and examined, and smaller diagnostic jobs can be run from them. Most important of all, code development or maintenance does not disrupt the rest of the system.

5. High Level Modeling Systems and NLP Code Developers.

The first sections of this paper described advantages that modelers could derive from a high level modeling system. This section briefly notes some of the potential benefits for researchers at the other end of the NLP world - the developers of NLP codes.

As the use of modeling systems increases and users with non-linear applications become more productive, the market for non-linear codes will increase. Much of this increase will come from users who have hitherto remained with linear formulations because of the high cost of converting existing data representations. A non-linear code that is accessed only through a modeling system is also smaller and less cumbersome than a traditional stand-alone code. Detailed and elaborate input data checking and attempts at recovery are not needed, nor are complex outpu routines. In MINOS 5.0 we were able to eliminate about 30% of the code because we use our own input routines, which we know will never encounter a syntactically incorrect problem representation. User documentation will also be much shorter and easier to write. The other big advantage to be obtained from a high level modeling system is the availability of a wide selection of test

problems of all sizes. These help not only to test the code under development, but also to compare it to other established codes. A standard library of 60 published models is already available for GAMS, and more are being added.

Interfaces to modeling systems are not always simple to build and their creation requires specialized expertise. During the coming year the GAMS team expects to define ways of standardizing the input to NLP codes with the goal of delivering the components of an easy 'do-it-yourself' approach to interfacing codes.

6. Experience with GAMS.

GAMS has been available for LP models since 1980, and for NLPs since 1982. It has been used in the World Bank and at collaborating research and teaching institutions by people with widely differing experience in mathematical programming, mostly for solving problems in policy analysis, economics, or engineering. The benefits to these users have been substantial. The modeler is certainly more productive. In many cases an economist has been able to avoid employing a programmer to translate algebra into the machine representation, thus maintaining direct control of the project while still producing timely results. Perhaps more importantly the details of a modeling effort are now more readily accessible to others, and the traditional question about 'what really happens in this model' can be answered.

GAMS has also been found very useful in teaching. Models set up and solved by students have typically been small, because understanding and implementing the data structure for complex models is a tedious and

specialized task. GAMS and its library of models permits students to concentrate on the model and not on the details of its implementation, and to experiment with formulation or data changes on models large enough to be interesting.

Finally, it is worth noting, some usage statistics. In the World Bank alone 15 projects currently use GAMS. Non-linear problems range up to 700 equations and now outnumber linear problems. The GAMS system is accessed by World Bank users about 50 times on a normal weekday.

APPENDIX 1: GAMS/NLP Interfaces

A. Interface with GRG2

This appendix describes the first GAMS/NLP interface that was developed, that to GRG2. As we saw in Table 1, GRG2 does not need much information about the problem, and to make the task even easier we decided to let GRG2 compute all derivatives. The task was therefore essentially to create a FORTRAN subroutine, called FCOMP, that for a given vector of variables could return a vector of constraint values. The conceptual components of this FORTRAN subroutine are:

1. Variable Mapping. GRG2 does not know that GAMS thinks of the problem in terms of different blocks of variables; it only knows a linear sequence of variables, X(1), X(2),.,,,X(N). Once this vector is received by FCOMP it must, at least conceptually, be decomposed into the blocks of variables that GAMS uses.

2. Constraint Evaluation. FCOMP must now compute the residuals of the equations. It is written with outer loops for the equations, and inner loops for SUMs and PRODs. Parameter values are picked up from a data area within FCOMP, and the values of the variables are picked up from the decomposed input vector. The execution itself is done in much the same way as in the execution system of GAMS.

3. Constraint Mapping. Save the residuals or constraint values in the output vector. Again, GAMS thinks of the constraints in terms of blocks but GRG2 only knows a linear sequence of constraints, G(1), G(2),..., G(M), and the results of step 2 must, at least conceptually, be mapped into G.

There are several difficulties involved in implementing the conceptual steps above. The key ones are:

1. The blocks of GAMS variables are naturally placed after each other in GRG2's X-vector. It is not enough, however, to define the start within X of each block of GAMSs variables and then to compute addresses within the blocks. GAMS variables need not be defined over the full cartesian product of its index sets. The variable IN(FA,SC) in the model in section 3 is an example; only five out of the possible six variables are actually in the model. FCOMP must therefore keep an explicit mapping from the cartesian product of the index spaces (from the six possible variables) into the actual positions in X (into the five variables actually used).

2. Some variables and parameters are indexed over one set in one part of the model and over a different set, e.g., a subset, somewhere else in the model. An example is PD in equation EXD above. GAMS must necessarily perform the index loops over the union of the sets because the variable mappings and the parameter addresses must have a global nature. Subsets must then be handled as exceptions to the large set. The only alternative would be to have different mappings for the same GAMS variable at different locations in the model.

Implementation can now be broken down into 5 steps:

1. Find the dimensions of the model. A dimension of a model is a set consisting of all elements that are related in one way or another. The model components described in section 3 have two dimensions, a sector dimension and a factor dimension. The dimensions are generally determined by observing the use of the basic sets of a model; whenever one set is used in the same index position as another set in a reference to a parameter or

variable, then the two sets must represent the same type of information and belong to the same dimension. Repeated use of this rule generates the dimensions.

2. Define all sets as subsets of their dimensions and create vectors of subset indicators. Replace all references to sets by references to subsets of the corresponding dimensions.

3. Set up the exogenous data for the model in multidimensional arrays defined over the cartesian products of the relevant dimensions.

4. Create the FCOMP subroutine and compile and load it with GRG2. Before we pass control to GRG2, execute FCOMP in special set up mode. This mode uses arrays of indicators for all variables and equations. The indicators are initially set to 'notused', and during the setup execution the indicators are set to 'used' each time a variable or an equation is referenced.

5. Squeeze all variables and equations marked 'notused' out of the model and create mapping arrays for variables and equations. Use these mappings in all later calls of FCOMP. Call GRG2 and constraint evaluations can begin.

The link back to GAMS is also straightforward. We have added a special binary output file to GRG2 containing both primal and dual information in much the same format as supplied by commercial LP systems. GAMS reads this file, maps the GRG2 variables and equations into GAMS variables and equations, and saves the solution in its database.

B. Interface with CONOPT

This section describes the components that had to be added to the GRG2 interface in order to handle the extra requirements associated with sparsity in CONOPT, and to implement 'squeezing' based on exogenous data values as well as the logical 'squeezing' used with GRG2.

The first change was to write two versions of FCOMP, a special setup version and a simpler execution time version. The setup version has extra code that detects which variables and equations are used and stores the equation/variable identifiers as a pair each time a variable is referenced. We simultaneously evaluate constant expressions and do not write entries if the coefficient is guaranteed to be zero. This information is sorted, duplicates are removed, and the Jacobian structure is ready for output in a MPS file.

The second addition had to do with detecting which Jacobian elements were linear and which were nonlinear. The internal representation of expressions in GAMS relies on a stack machine, and it is fairly easy to analyze linearity within this framework. Each operand on the stack is classified as constant, linear, or nonlinear. When an exogenous value is pushed on the stack it will be classified as constant, and when a variable is pushed on the stack it will be classified as linear. When an operation is applied to one or two operands on the stack, the result will be classified on the basis of the operation and classification of the operands. Addition of a constant and a linear operator will create a linear operator, and multiplication of two linear operators will create a nonlinear operator. Whenever an operand on the stack becomes nonlinear, all variables that

contributed to this operand are classified as nonlinear variables in this equation, and the corresponding Jacobian elements are nonlinear.

The analysis of which Jacobian elements are nonlinear can be done on a block level, i.e., we can determine whether IN(*,*) appears nonlinearly in PRDF(*), or on an individual equation level, i.e. we can determine whether IN('LABOR','SERVICES') appears nonlinearly in PRDF('SERVICES'). The block approach is much faster since most models have several equations in each block of equations and several variables in each block of variables. This approach may, however, classify a few linear elements as nonlinear. An example is IN('LABOR','SERVICES') in equation PRDF('SERVICES'). The power is exactly one in this case, so the term is linear, even though the overall block must be classified as nonlinear. The CONOPT interface uses a block approach and this information is needed to generate the two FCOMPS.

C. Interface with NPSOL

This section describes the features that are unique to the NPSOL interface, i.e., the single equation approach and the derivatives.

The single equation approach is quite simple. GAMS loops over the indices of the constraints and creates a separate constraint for each combination. All loops within each constraint, e.g., loops associated with SUMs and PRODs, are unrolled, all computations that only depend on parameter values are performed, and products with zeros, logarithms of one, expressions to the power one, and all exceptions are used to remove terms. The result is a constraint with references to variables with known indices and references to constants, tied together by simple operators only.

The derivatives are based on this simplified constraint expression. As described in Section B of this Appendix, GAMS uses a stack oriented representation of the constraint expressions. The derivatives are computed in parallel to the function values. For each operator on the stack, GAMS remembers the numerical value of its derivatives with respect to all variables, and it knows which of them are zero, one, another constant or variable. When a constant is pushed on the stack all derivatives of the operator are zero. When a variable is pushed on the stack all derivatives are zero except the derivative with regard to the variable that was pushed; this derivative is 1. When an operation is applied to one or two operands on top of the stack all derivative are computed using the chain rule. The derivative of a sum is the sum of the derivatives, and the derivative of a product, e.g., u*v, is u'*v+u*v', where ' denotes derivative. Since GAMS knows which derivatives are zero and one it can eliminate additions of zero and products of one; the resulting set of nontrivial instructions created by GAMS is quite compact.

The internal GAMS code was converted into a FORTRAN subroutine for GRG2 and CONOPT. Here we have chosen to write the code into a file and interpret it in the model subroutine. The subroutine is model independent and we can therefore create an absolute program with the optimizer and the interpreter, thereby saving such fixed costs as those of creating FORTRAN code, compiling it, and linking it to the optimizer itself. We do extensive work preprocessing the code to be interpreted, including hardwiring of all addresses or indices, and the speed of the interpreter has turned out to be impressive. Most optimizations use less than 10 percent of the overall optimization time in the interpreter, and a model has to be very complicated before the time ratio exceeds 40 percent.

REFERENCES

[1] J. BISSCHOP and A. MEERAUS, "On the development of a General Algebraic Modeling System in a Strategic Planning Environment", Mathematical Programming Study, 20 (1982) 1-29.

[2] M. BROWN, A. DAMMERT, A. MEERAUS, and A. STOUTJESDIJK, "Worldwide Investment Analysis, The Case of Aluminum", World Bank Staff Working Paper, 603 (1983) 168p.

[3] A. M. CHOKSI, A. MEERAUS, and A. STOUTJESDIJK "The Planning of Investment Programs in the Fertilizer Industry", John Hopkins University Press, (1980) 333p.

[4] C. F. COLEMAN and J. J. MORE, "Estimation of Sparse Jacobian Matrices and Graph Coloring Problems", SIAM Journal of Numerical Analysis, 20 (1983) 187-209.

[5] A. DRUD and A. MEERAUS, "CONOPT, A System for Large Scale Dynamic Nonlinear Optimization, User's Manual", Technical Note 16, Development Research Center, World Bank, (1980) 70p.

[6] A. DRUD, "Interphasing new solution algorithms with existing modeling systems" Journal of Economic Dynamics and Control, 5 (1981) 131-149.

[7] A. DRUD, "CONOPT - A GRG Code for Large Sparse Dynamic Nonlinear Optimization Problems", Mathematical Programming, (forthcoming).

[8] S. I. GASS, K. L. HOFFMAN, R. H. F. JACKSON, L. S. JOEL, and P. B. SAUNDERS "Documentation for a Model: A Hierarchical Approach" Communications of the ACM 24 (1982) 728-733.

[9] E. P. GILL, W. MURRAY, M. A. SAUNDERS, and M. H. WRIGHT, "User's Guide for SOL/NPSOL: A Fortran Package for Nonlinear Programming", Department of Operations Research, Stanford University, (1984) 34p.

[10] D. KENDRICK, A. MEERAUS, and J. ALATORRE, "The Planning of Investment Programs in the Steel Industry", John Hopkins University Press, (1984) 310p.

[11] D. KENDRICK and A. MEERAUS, "GAMS, An Introduction", mimeo, Development Research Department, World Bank, (1984).

[12] L. S. LASDON, A. D. WAREN, A. JAIN, and M. RATNER, "Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming", ACM Transactions on Mathematical Software, 4 (1978) 34-50.

[13] A. MEERAUS, "General Algebraic Modeling System (GAMS), User's Guide version 1.0", Development Research Center, World Bank, (1982).

[14] A. MEERAUS, "An Algebraic Approach to Modeling", Journal of Economic Dynamics and Control, 5 (1983) 81-108.

[15] B. A. MURTAGH and M. A. SAUNDERS, "A Projected Lagrangian Algorithm and its Implementation for Sparse Nonlinear Constraints", Mathematical Programming Study 16 (1982) 84-117.

[16] B. A. MURTAGH and M. A. SAUNDERS, "MINOS 5.0 User's Guide", Technical Report SOL 83-20, Department of Operations Research, Stanford University, (1983) 114p.

[17] K. SCHITTKOWSKI, "Nonlinear Programming Codes", Springer-Verlag Lecture Notes in Economics and Mathematical Systems, 183 (1980).